(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

Co

THESIS

A CONSTRAINT BRANCH-AND-BOUND METHOD FOR SET PARTITIONING PROBLEMS

by

Moo Bong Ryoo

March, 1990

Thesis Advisor: R. Kevin Wood

Approved for public release; distribution is unlimited.

Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified			1b Restrictive Markings
2a Security Classification Authority			3 Distribution Availability of Report
2b Declassification Downgrading Schedule			Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(if applicable)* 55		7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000			7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding Sponsoring Organization	8b Office Symbol *(if applicable)*		9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*			10 Source of Funding Numbers
		Program Element No	Project No / Task No / Work Unit Accession No

11 Title *(include security classification)* A CONSTRAINT BRANCH-AND-BOUND METHOD FOR SET PARTITIONING PROBLEMS

12 Personal Author(s) Moo Bong Ryoo

13a Type of Report	13b Time Covered	14 Date of Report *(year, month, day)*	15 Page Count
Master's Thesis	From To	March 1990	47

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Set partitionig problem, Constraint branch and bound method, Enumeration tree.

19 Abstract *(continue on reverse if necessary and identify by block number)*

This thesis compares the efficiency of a constraint branch-and-bound method against the conventional variable branch-and-bound method in solving set partitioning problems. Because of the difficulties encountered in writing the constraint branch-and-bound subroutine, it was necessary to solve each subproblem encountered from scratch. This is in contrast to the variable branching code which, when solving closely related subproblems, essentially starts from an advanced starting solution. Even using an inefficient implementation, the constraint branch-and-bound method appears to be significantly more efficient than the conventional variable branch-and-bound method. It saves, on average, 30.0 % in CPU time over the variable branch-and-bound method when tested on a set of small test problems. On average, constraint branch and bound produces 59.3 % fewer nodes in its enumeration trees than does variable branch and bound, and the trees encountered are shallower and better balanced.

20 Distribution Availability of Abstract	21 Abstract Security Classification	
☒ unclassified unlimited ☐ same as report ☐ DTIC users	Unclassified	
22a Name of Responsible Individual R. Kevin Wood	22b Telephone *(include Area code)* (408) 646-2768	22c Office Symbol 55Wd

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

A Constraint Branch-and-Bound Method
For Set Partitioning Problems

by

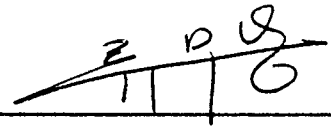
Moo Bong Ryoo
Captain, Republic of Korea Army
B.S., Korea Military Academy, 1986

Submitted in partial fulfillment of the
requirements for the degree of

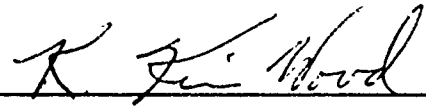
MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
March 1990

Author: 

Moo Bong Ryoo

Approved by: 

R. Kevin Wood, Thesis Advisor

Gerald Gerard Brown, Second Reader

Peter Purdue, Chairman,
Department of Operations Research

ABSTRACT

This thesis compares the efficiency of a constraint branch-and-bound method against the conventional variable branch-and-bound method in solving set partitioning problems. Because of the difficulties encountered in writing the constraint branch-and-bound subroutine, it was necessary to solve each subproblem encountered from scratch. This is in contrast to the variable branching code which, when solving closely related subproblems, essentially starts from an advanced starting solution. Even using an inefficient implementation, the constraint branch-and-bound method appears to be significantly more efficient than the conventional variable branch-and-bound method. It saves, on average, 30.0 % in CPU time over the variable branch-and-bound method when tested on a set of small test problems. On average, constraint branch and bound produces 59.3 % fewer nodes in its enumeration trees than does variable branch and bound, and the trees encountered are shallower and better balanced.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my thesis advisor, Professor R. Kevin Wood. He was a consistent wellspring of sound advice, technical competence, professional assistance, and moral support. Without his assistance, this thesis could not have been undertaken.

I am also thankful to Professor Gerald G. Brown who taught me about linear programming, integer programming, and provided the idea for this thesis.

I am also like to thank to my fellow Korean students for their encouragement, and my other classmates who have helped this foreign student a lot.

Now it is time to go home and I'd like to thank my family, far away in Korea, who have stayed in my heart all the way through.

I. INTRODUCTION

Many scheduling and routing problems arising in the real world can be posed as large Set Partitioning Problems (SPPs). Most solution techniques for such Integer Linear Programs (ILPs) are based on solving the Linear Programming (LP) relaxation by relaxing the integer requirement of the problem and then resolving fractional variables, if any occur. The technique of branch and bound (sometimes coupled with other techniques) has proven to be the most reliable method of resolving fractional variables.

There are two basic branching methods, variable branching and constraint branching. Conventional variable branching chooses a fractional variable x_j in the solution of the LP relaxation and forces that variable to 0 or 1 by effectively adding the constraint $x_j = 0$ or $x_j = 1$. The one-branch (setting $x_j = 1$) tends to increase the minimized objective value significantly since, typically, x_j occurs in several constraints along with many other variables and all those other variables are effectively set to 0. The zero-branch (setting $x_j = 0$) fixes only one variable and tends to lead to a similar fractional solution. Typically, the objective value increases only slightly, or not at all. Thus, the variable branch-and-bound enumeration tree tends to develop unevenly which can lead to slow convergence.

The constraint-branching method may offer a way of developing more balanced and smaller branch-and-bound trees. The method selects a pair of constraints (i, k) in the current solution of the LP relaxation which have at least two fractional variables in common. Let $J(i, k)$ denote the index set of variables which the rows i and k have in common. Then, a branching consists of enforcing either $\Sigma_{j \in J(i,k)} x_j = 1$ or $\Sigma_{j \in J(i,k)} x_j = 0$. Unlike variable branching, the zero-branch here sets at least two variables to 0. Assuming constraints i and k have at least two variables not in common, the one-branch

also effectively sets at least two variables to 0. Thus, it appears that constraint branching may yield a more balanced, and possibly smaller branch-and-bound tree. Furthermore, it will be seen that explicit constraints need not be added to the LP to implement constraint branching in SPPs.

The main purpose of this thesis is to examine the efficiency of constraint-branching methods and compare them to conventional variable branching in SPPs. Earlier work on such comparisons is limited and the test problems are very specialized. In this work, branch-and-bound algorithms are implemented in FORTRAN and embedded within a primal-dual simplex solver. A set of artificial test problems is used for comparing the efficiency of the two methods. A secondary purpose of this thesis is to state an algorithm for constraint branch and bound in a concise and compelling fashion. This is done because the primary references for the constraint branch-and-bound method are diffuse and do not individually include precise statements of their algorithms.

A. SET PARTITIONING PROBLEMS

The following definition of a set partitioning problem is taken from Garfinkel and Nemhauser [Ref. 1]. Consider a set $I = \{1, 2, ..., m\}$, and a set $P = \{P_1, ..., P_n\}$, where $P_j \subseteq I$, $j \in J = \{1, ..., n\}$. A subset $J' \subseteq J$ defines a partition of I, if the following two conditions hold:

$$\bigcup_{j \in J^*} P_j = I$$

and

$$j, k \in J^*, \quad j \neq k \Rightarrow P_j \cap P_k = \emptyset.$$

If a cost c_j is associated with every $j \in J$, then the total cost of the partition J^* is $\sum_{j \in J^*} c_j$. The SPP is to find a partition J^* which has minimum cost.

The SPP can be written in standard mathematical programming notation as

$$\text{Minimize} \quad z = \sum_{j=1}^{n} c_j x_j$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} x_j = 1, \qquad i = 1, 2, ..., m$$

$$x_j \in \{0, 1\} \qquad j = 1, 2, ..., n$$

where all the a_{ij} are 0 or 1. Specifically, $a_{ij} = 1$ if subset P_j contains element i, and 0 otherwise.

SPPs typically arise in scheduling and routing problems. For instance, the set I might consist of a set of deliveries which must be made. Each $j \in J$ corresponds to a tentative delivery route, say a day's worth of deliveries for a truck, such that $a_{ij} = 1$ if delivery i is made on route j, and 0 otherwise. A solution is the partition of deliveries into routes. SPPs (and set covering problems where $P_j \cap P_k = \emptyset$ is not necessary) have been studied widely because of their many applications and because of their intriguing binary structure. Some of the applications of SPPs which have appeared in the literature include air crew scheduling by Arabeyre, Fearnley, Steiger, and Teather [Ref. 2], and Marsten and Shepardson [Ref. 3], truck routing by Clarke and Wright [Ref. 4], political districting by Garfinkel and Nemhauser [Ref. 5], vehicle scheduling by Foster and Ryan [Ref. 6], and bus driver scheduling by Smith [Ref. 7].

Three basic methods have been proposed to solve SPPs: cutting plane, branch and bound and implicit enumeration. The cutting-plane methods (e.g., Balas and Padberg,

[Ref. 8]) start by solving the LP relaxation of the SPP, and then add additional constraints in an attempt to cut away noninteger solutions. Unfortunately, in practice, adding a cut typically increases the number of fractional variables which makes succeeding cuts more difficult to generate and enforce. Furthermore, cutting-plane methods guarantee an integer solution only in theory.

The implicit enumeration method (e.g., Etcheberry, [Ref. 9]) may be thought of as branch and bound in which no LPs need to be solved. In practice, poor bounds on the optimal solution are available and using these leads to an enormous amount of enumeration. Consequently, implicit enumeration is only suitable for very small problems or problems with special structure.

Branch and bound (e.g., Garfinkel and Nemhauser[Ref. 1], Davis et al.[Ref. 10], Little, et al.[Ref. 11]) has proven to be the most reliable method for solving set partitioning and other integer programming problems. Conventional variable branch-and-bound methods start by solving the LP relaxation of the ILP, and then choose a fractional variable x_j in the current solution, and set that variable to 1 or 0 by effectively adding the constraint $x_j = 1$ or $x_j = 0$. Branch-and-bound methods guarantee an integer solution if it exists by including one integer variable at each branch. In addition to variable branch and bound, in certain kinds of problems, constraint branching is also possible.

B. CONSTRAINT BRANCHING

Marsten [Ref. 3] developed a constraint-branching technique for SPPs, and Hey [Ref. 12] applied a constraint-branching method to set covering problems. The method selects a pair of constraints (i, k) in the current relaxed LP solution, which have at least two fractional variables in common. The branch can be either $\Sigma_{j \in J(i,k)} x_j = 1$ or $\Sigma_{j \in J(i,k)} x_j = 0$. Let $J'(i, k)$ be the set of variables which cover either constraint i or j, but not both. Then, the one-branch, setting $\Sigma_{j \in J(i,k)} x_j = 1$ is the same as setting

$\Sigma_{j \in J'(i,k)} x_j = 0$. But, $\Sigma_{j \in J'(i,k)} x_j = 0$ can be implemented by setting $x_j = 0$ for all $j \in J'(i, k)$. Thus no new explicit constraint need be added. The zero-branch, $\Sigma_{j \in J(i,k)} x_j = 0$, is equivalent to $x_j = 0$ for all $j \in J(i, k)$, and once again no new, explicit constraint need be added. Both the zero-branch and one-branch fix at least two variables to 0, if it is assumed that there are at least two variables in sets $J(i, k)$ and $J'(i, k)$. This should lead to a more balanced tree than in the variable-branching method, and may result in a smaller branch-and-bound tree. Falkner and Ryan [Ref. 13] successfully applied constraint-branching techniques to solve a bus driver scheduling problem.

C. THESIS SCOPE

The variable-branching methods and constraint-branching methods and their algorithms are reviewed in general in Chapter II. The actual implementation of constraint branching by fixing a set of variables to 0, and a releasing set of variables is also described there. Chapter III gives computational results for both constraint branching and variable branching on a set of artificial test problems.

II. ALGORITHMS FOR VARIABLE BRANCH AND BOUND AND CONSTRAINT BRANCH AND BOUND

A. INTRODUCTION

Branch and bound is an optimization technique that uses basic tree enumeration (Garfinkel and Nemhauser [Ref. 1]). It involves calculating an upper bound and a lower bound on the objective function in order to accelerate the fathoming process. A node g (the LP relaxation plus additional restriction) is said to be fathomed if no more exploration (further restriction) can be profitable from that node. Node g can be fathomed in the following three cases. In the first case, fathoming occurs when an integer solution is found at node g. Second, it can occur by bounding on the best known solution to the problem. In the first and second case, no successor (further restriction) of node g can yield a solution that improves on the best known solution. The third case occurs when a node corresponds to an infeasible problem.

The term ***branch and bound*** was first used by Little et al [Ref. 11]. The branch-and-bound method was developed first as a method based on fixing binary variables to 0 or 1. Later, Marsten [Ref. 14] developed a constraint-branching algorithm for SPPs. He suggested that the basic choice in a constraint branch should not be based on whether a specific variable covers a particular row or not. Instead the variables are grouped as sets, and the basic choice involved is in choosing which set should be responsible for covering a particular row.

In Section B, the basic algorithm for the variable-branching method is reviewed. In Section C, the general concept and algorithm for the constraint branching method is discussed. A more detailed description of an actual implementation of the constraint-branching method for SPPs follows.

B. VARIABLE BRANCH-AND-BOUND METHODS

The variable branch-and-bound method for solving ILPs is based on fixing variables. This thesis limits itself to considering only binary ILPs, because binary variables are the most common integer variables and because the ultimate aim is to consider SPPs, which involve only binary variables. When the relaxed LP solution contains one or more fractional variables in the solution, the variable branch-and-bound method selects a variable and fixes it to 0 or 1 (Hey [Ref. 12] summarizes three ways of choosing a desirable variable in set covering problem). The method then solves a new LP relaxation with the added restriction, and executes the variable selection procedure until the current node gets fathomed. The following is an algorithm for minimizing the objective function value. The algorithm uses a depth-first strategy for exploring the enumeration tree.

A depth-first variable branch-and-bound algorithm

- STEP 1: (Initialization) Let the stack S be empty, $\bar{z} = \infty$ (upper bound).
- STEP 2: Subject to the constraints defined with respect to S, solve the LP relaxation of the ILP, to obtain objective value z. Go to step 3.
- STEP 3: (Fathoming test) If an integer solution is found at step 2, or $z \geq \bar{z}$, or the problem becomes infeasible, go to step 5. Otherwise go to step 4.
- STEP 4: (Branching) Select a fractional variable and fix it to 0 or 1 using some selection and branching criteria. Push that variable on S, and mark it as 'not reversed'. Go to step 2.
- STEP 5: (Fathoming) If fathoming occurs by finding an integer solution, and if $z < \bar{z}$, then let $\bar{z} = z$, and save the current solution. If S is empty, go to step 8. Otherwise, go to step 6.
- STEP 6: (Reversing) Consider the variable on the top of S, and check if it has been reversed. If it is not reversed, mark it as 'reversed', fix the variable to its opposite bound, and go to step 2. Otherwise go to step 7.
- STEP 7: (Backtracking) Pop the top variable from S and free the variable. If S is empty, go to step 8, else go to step 6.
- STEP 8: (Termination) If $\bar{z} = \infty$, there is no feasible solution, otherwise $\bar{z}$ is the optimal objective value and the last solution saved is the optimal solution.

C. CONSTRAINT BRANCH-AND-BOUND METHODS

In SPPs, let i and k denote constraint indices, and let j represents a variable index. The notation $J(i)$ denotes the set $\{j \mid a_{ij} = 1\}$, while $J(i, k)$ denotes the set $J(i) \bigcap J(k)$, and $J'(i, k)$ denotes $\{ J(i) \bigcup J(k) \} - J(i, k)$. For constraint-branching, a ***one-branch*** corresponds to setting the sum of a set of variables to 1, say, $\Sigma_{j \in J(i,k)} x_j = 1$. A ***zero-branch*** sets the sum of variables to 0, such as $\Sigma_{j \in J(i,k)} x_j = 0$.

When the relaxed LP solution of an SPP contains fractional variables, the basic choice of the constraint branch-and-bound method is not whether a certain variable should cover a particular constraint or not. Instead, the variables are grouped into classes and the basic choice decides whether or not a certain class of variables should be responsible for covering a particular constraint. The status of individual variables is determined automatically. The simplest constraint branch can be written

$$\sum_{j \in \hat{J}} x_j = 0 \quad \text{or} \quad \sum_{j \in \hat{J}} x_j \geq 1$$

where $\hat{J}$, a subset of $J = \{ 1, 2, ..., n \}$ is an index set such that $0 < \Sigma_{j \in \hat{J}} x_j < 1$ in the current relaxed solution. In practice, $a_{ij} = 1$ for all $j \in \hat{J}$ for some i and thus the one-branch implies that set $\hat{J}$ will be responsible for covering row i and the zero-branch implies it will not. Furthermore, the constraint $\sum_{j \in \hat{J}} x_j \geq 1$ becomes $\sum_{j \in \hat{J}} x_j = 1$. The method then solves a new relaxed but constrained LP again, and executes the constraint-branching procedure until all vertices are fathomed. The following is an algorithm, using the depth-first-search strategy, for a general constraint branch-and-bound method. The algorithm minimizes the objective function.

A depth-first constraint branch-and-bound algorithm

- STEP 1: (Initialization) Let $\bar{z} = \infty$, and let the stacks S and T be empty.

- STEP 2: Subject to the constraints defined with respect to S and T, solve the LP relaxation of the ILP to obtain objective value z.
- STEP 3: (Fathoming test) If an integer solution is found at step 2, or $z \geq \bar{z}$, or the problem becomes infeasible, go to step 5. Otherwise go to step 4.
- STEP 4: (Branching) Using some selection criteria, select a set of variables in $\hat{J} \subseteq J(i)$, for some i, whose sum is strictly between 0 and 1 in the current solution, and using some branching criteria, enforce the constraint $\Sigma_{j \in J} x_j = 1$ or $\Sigma_{j \in J} x_j = 0$. Push fixed variable indices in $\hat{J}$ on S, and push the pointers for the starting and ending points of that set on T, and mark the two pointers as 'not reversed'. Go to step 2.
- STEP 5: (Fathoming) If fathoming occurred by finding an integer solution, and if $z < \bar{z}$ then let $\bar{z} = z$, and save the current solution. If S is empty, go to step 8, else go to step 6.
- STEP 6: (Reversing) Consider the two pointers on the top of T, and check if they marked as 'reversed'. If they are marked as 'reversed', go to step 7. Otherwise reverse the equality defined by the set $\hat{J}$ on the top of S, and mark the two pointers on the top of T as 'reversed'. Go to step 2.
- STEP 7: (Backtracking) Pop the two pointers from the top of T, and pop the variables from the top of S using the pointers. If S is empty, go to step 8. Otherwise go to step 6.
- STEP 8: (Termination) If $\bar{z} = \infty$, there is no feasible solution. Otherwise $\bar{z}$ is the optimal objective value and the last solution saved is the optimal solution.

D. IMPLEMENTATION OF CONSTRAINT BRANCH AND BOUND IN SET PARTITIONING PROBLEMS

The constraint branch-and-bound method for SPPs was first developed by Marsten [Ref. 14]. The constraint matrix $A = a_{ij}$ is first placed in staircase form. Then let the rows be fixed and define B_i as the set of all those columns which have their first nonzero coefficient in row i ($1 \leq i \leq m$). The algorithm then assigns each rows k to a block B_i, rather than a particular column. Assigning row k to B_i ($i < k$) implies the restriction that row i and k must be covered by the same column. Thus row i and k are covered by the variables which are in both rows i and k, and are not set to 0. Marsten assigns the rows in increasing order of row index, and performs logical tests to ensure that the assignment of a row to a block is feasible. The branch is implemented by successively

banning variables (setting variables to 0) which are not assigned to a block from the solution.

Ryan and Falkner [Ref. 13] also develop a constraint branch-and-bound method for SPPs. On the one-branch, the sum of the variables which cover both i and k are set to 1 ($\Sigma_{j \in J(i,k)} x_j = 1$), which is implemented as $x_j = 0 \ \forall \ j \in J'(i, k)$. On the zero-branch, $\Sigma_{j \in J(i,k)} x_j = 0$, which is implemented as $x_j = 0 \ \forall \ j \in J(i, k)$. Ryan and Falkner choose a pair of constraints (i, k) which has $\Sigma_{j \in J(i,k)} x_j$ fractional, and then require that sum to be 0 or 1, which makes the current solution infeasible.

In this thesis the Ryan and Falkner method [Ref. 13] is adopted. In the actual implementation of constraints branch-and-bound step, the following approach is used.

Choice of Node : A depth-first-search strategy is used to select the next node to explore.

Choice of Branch : The one-branch is always explored first.

Choice of a Pair of Constraints : The first pair of constraints (i, k) which has $0.4 \leq \Sigma_{j \in J(i,k)} x_j \leq 0.6$, and $| J(i, k) | > 2$ is selected. If no such pair exists, then the pair of constraints (i, k) which has $\Sigma_{j \in J(i,k)} x_j$ closest to 0.5 is selected.

Data Structures : Two stacks S and T are used, but in a different manner than described in Section C. Stack S contains the set of variables which are currently fixed to 0. The first variable in a set such as $J(i, k)$ is marked by a negative sign which allows easy removal of all variables in a set without a special stack of pointers. Stack T contains the set of constraint pairs (i, k) for each of the sets on S. When branching to the one-branch, the status of the variables is noted using a special array STATUS. STATUS (j) = ***fixed/not reversed*** indicates that the upper and lower bounds on x_j are 0 (x_j has been fixed to 0), but the current branch in which j is involved is a one-branch. STATUS (j) = ***fixed/reversed*** also indicates that the upper and lower bounds on x_j are 0 but the current branch in which j is involved is a zero-branch. Two other states are also indi-

cated by STATUS. STATUS $(j) =$ ***free*** indicates that x_j may take on any value between 0 and 1. STATUS $(j) =$ ***to be freed*** indicates that x_j is fixed to 0 but about to become ***free.***

The following is an algorithm of constraint branch and bound used in this thesis for set partitioning problems. The algorithm minimizes the objective function.

A depth-first constraint branch-and-bound algorithm for set partitioning problems

- STEP 1: (Initialization) Let $\bar{z} = \infty$, let the stack S and T be empty and let STATUS $(j) =$ ***free*** $\forall j \in J$.
- STEP 2: Subject to the constraints defined with respect to S and T, solve the LP relaxation of the ILP to obtain objective value z.
- STEP 3: (Fathoming test) If integer solution is found at step 2, or $z \geq \bar{z}$, or the problem becomes infeasible, go to step 5. Otherwise go to step 4.
- STEP 4: (One-branch) Using the criteria described in Section D, select a pair of constraints (i, k). Set $x_j = 0$ and STATUS$(j) =$ ***fixed/not reversed*** for all j such that $j \in J'(i, k)$ and STATUS$(j) =$ ***free.*** Push indices $j \in J'(i,k)$ on S, with a negative sign on the first index in the set. Push the constraint indices i and k on T.
- STEP 5: (Fathoming) If fathoming occurs by finding an integer solution, and if $z < \bar{z}$ then let $\bar{z} = z$, and save the current solution. If S is empty, go to step 8, else go to step 6.
- STEP 6: (Reverse to zero-branch) Consider the variable j on the top of S, and if STATUS $(j) =$ ***fixed/reversed*** , go to step 7. Otherwise set STATUS $(j) =$ ***to-be-freed*** for the variables on S up to and including the first variable which has negative sign on its index. Pop the variables which have STATUS$(j) =$***to-be-freed.*** Fix $x_j = 0$, i.e., STATUS$(j) =$ ***fixed/reversed*** for all j such that $j \in J(i)$ and STATUS $(j) =$ ***free*** (This effectively fixes all unfixed variable in $J(i,k)$ to 0). Push the fixed variable indices on S with a negative sign on the first index in that set. For all $j \in J(i) \cup J(k)$ such that STATUS$(j) =$ ***to-be-freed,*** free j by setting STATUS $(j) =$ ***free.*** Go to step 2.
- STEP 7: (Backtracking) Pop the two constraint indices from the top of T, and pop the indices from the top of S up to and including the first index which has negative sign. While popping the variable indices j from the stack, free the associate variables by setting STATUS $(j) =$ ***free.*** If S is empty, go to step 8. Otherwise go to step 6.
- STEP 8: (Termination) If $\bar{z} = \infty$, there is no feasible solution. Otherwise $\bar{z}$ is the optimal objective value and the last solution saved is the optimal solution.

III. TEST RESULTS AND CONCLUSIONS

The FORTRAN program for the constraint branch-and-bound method was written in VS FORTRAN 77 and tested at the Naval Postgraduate School on an IBM 3033 AP computer operating under the CMS operating system.

The efficiency of the constraint branch-and-bound method with respect to the conventional variable branch-and-bound method is compared in terms of the number of nodes in the branch-and-bound tree, the maximum depths (number of node restrictions active at any one time) reached and the solution times. Eight sample problems are constructed, and used to compare the efficiency of the two techniques. The sample problems are summarized in Table 1, and the sample problems are listed in Appendix A. The results of the test runs are given in Table 2, and the enumeration tree of two sample problems are given in Figures 1 through 4.

A. COMPUTING AND PROGRAMMING ENVIRONMENT

The basic solver used for computational testing is the X-system [Ref. 15] which is a FORTRAN-based optimization system for linear, integer and nonlinear programming problems. The built-in variable-branching routine was used for all variable-branching testing. A specialized constraint-branching subroutine replaced the standard variable-branching subroutine for all constraint-branching tests. All programs are written in FORTRAN 77 and compiled with VS FORTRAN 2.3 at OPT(3). Input of the eight test problems was accomplished using a LINDO-type front end. Problems were run interactively in the CMS operating system. Because of difficulties encountered in writing the constraint branch-and-bound subroutine, it was necessary to solve each subproblem encountered from scratch. This is in contrast to the variable-branching code which, when solving closely related subproblems essentially starts from an advanced

starting solution. This typically leads to quicker solutions than starting from scratch does. Thus, while reporting solution statistics, time in CPU seconds is included, it should be noted that a significant improvement in solution speeds with constraint branching should be possible in the future. A better measure of computational efficiency for these tests is the number of nodes visited while branching.

B. SAMPLE SET PARTITIONING PROBLEMS

To test the efficiency of the constraint branch-and-bound method, 8 sample problems are constructed. Statistics on the size and density of these problems are given in Table 1. The sample problems have a maximum of 65 variables and 20 constraints which is quite small compared to real-world set partitioning problems. However, given the fact that standard branch and bound run time tends to explode exponentially as problem sizes increase, if an improvement is seen with constraint branch and bound on these small test problems this should mean even greater improvements are possible in full-scale problems. All of the sample problems were tested to ensure that they have optimal integer solutions. Sample problem JUL is taken from Falkner's thesis [Ref. 16], and modified slightly to ensure that some branching is necessary to solve the problem. Sample problem AIR is a crew scheduling problem taken from the LINDO manual

Table 1. SUMMARY STATISTICS FOR SAMPLE PROBLEM

Problem	constraints	variables	density
JUL	8	29	0.349
AIR	13	37	0.204
DON	20	40	0.196
T12	15	35	0.269
D3	20	45	0.214
SPD2X	20	60	0.116
D3X	20	65	0.141
D4	20	45	0.190

[Ref. 17], and also modified to require some branching. All other problems are devised by the author.

C. TEST RUN METHODOLOGY

The sample problems are solved 10 times with each method, and the smallest CPU time recorded to solve each problem with each method. This is done because there is some imprecision in timing in an interactive compute environment. The number of nodes in and depth of the branch-and-bound tree is obtained from output files. Also, the structure of the enumeration tree can be constructed using these output file. Table 2 gives the solution statistics for the two methodologies.

Table 2. SUMMARY TABLE OF TEST RESULTS

Sample	Constraint B&B			Variable B&B		
	time (seconds)	nodes	max depth	time (seconds)	nodes	max depth
JUL	0.41	5	2	0.55	15	4
AIR	0.78	7	3	0.69	17	4
DON	0.85	5	2	1.90	15	6
T12	1.41	11	3	2.60	35	10
D3	1.61	9	3	1.83	13	6
SPD2X	1.22	5	2	2.16	27	5
D3X	1.19	7	3	2.77	23	6
D4	2.89	17	5	3.37	25	10
Average	1.295	8.25	2.875	1.984	21.25	6.375

As it can be seen from the summary table of test runs, in all cases the constraint branch-and-bound method required significantly fewer nodes than did the variable branch-and-bound method. Also, the maximum depth reached was always less for constraint branch and bound. Finally, in all but one case constraint branch and bound

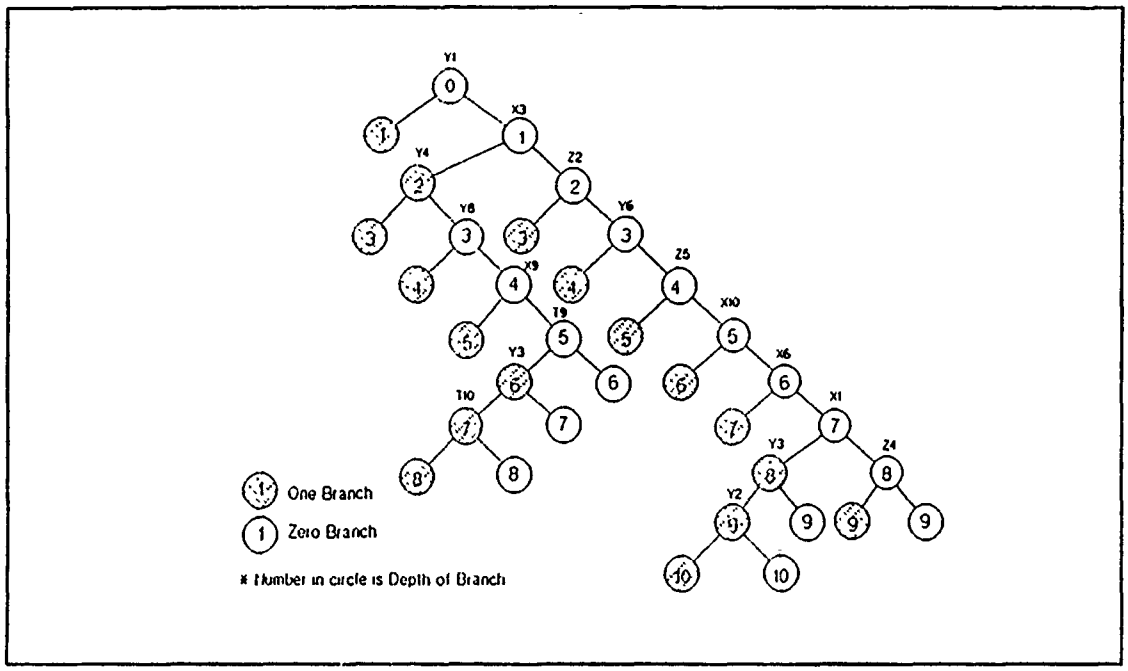


Figure 1. Variable enumeration tree for T12

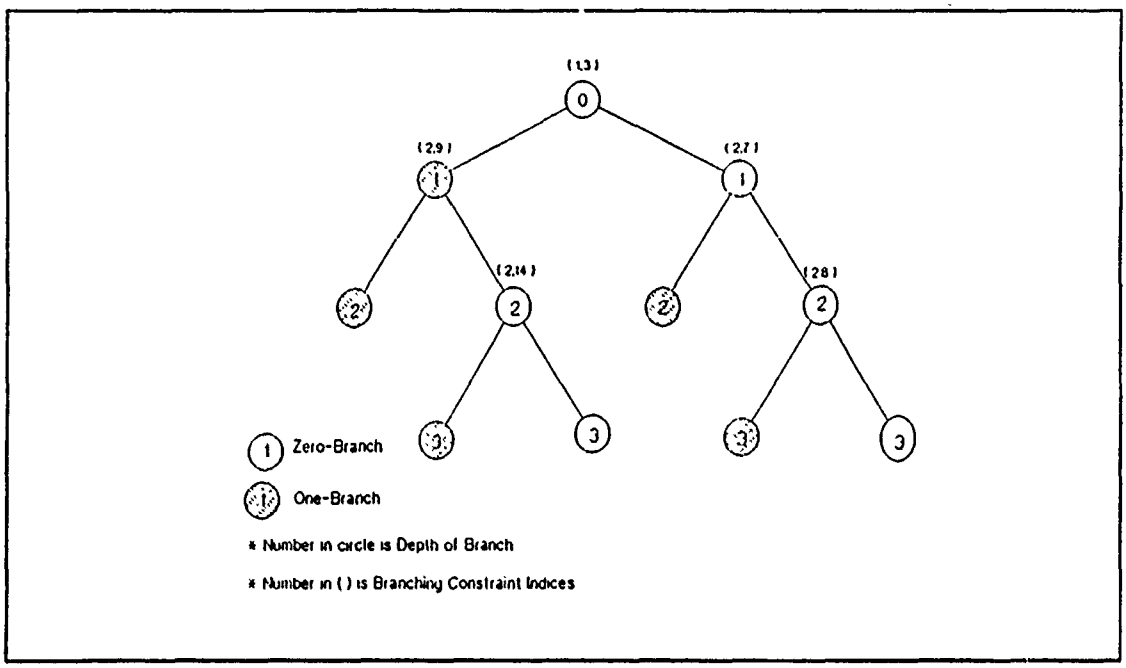


Figure 2. Constraint enumeration tree for T12

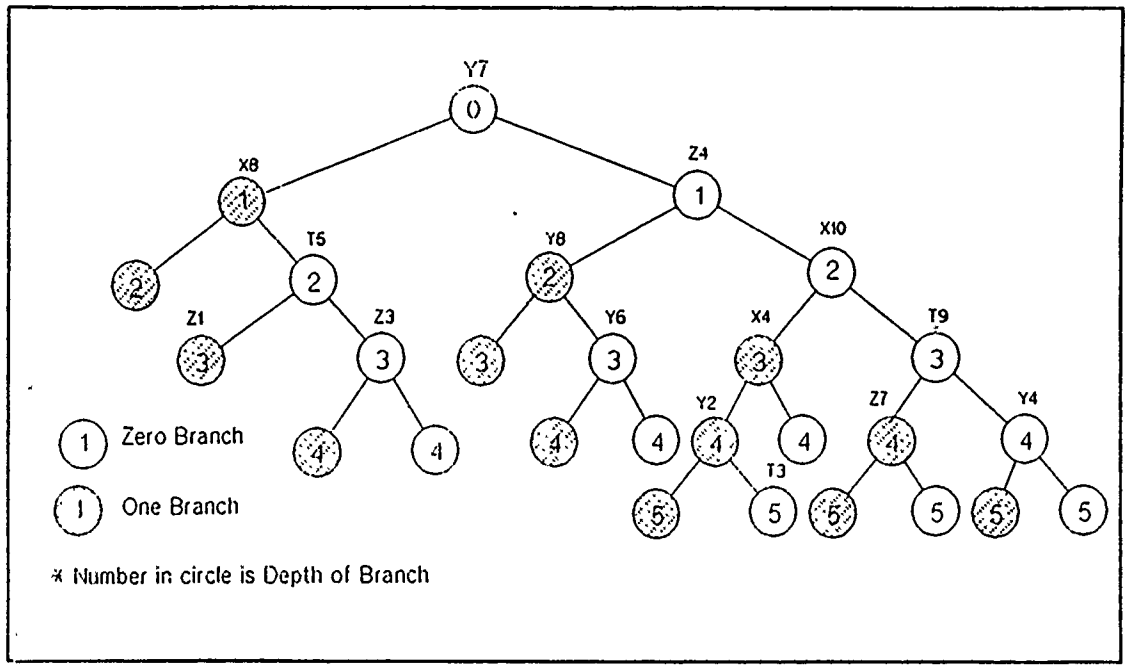


Figure 3. Variable enumeration tree for SPD2X

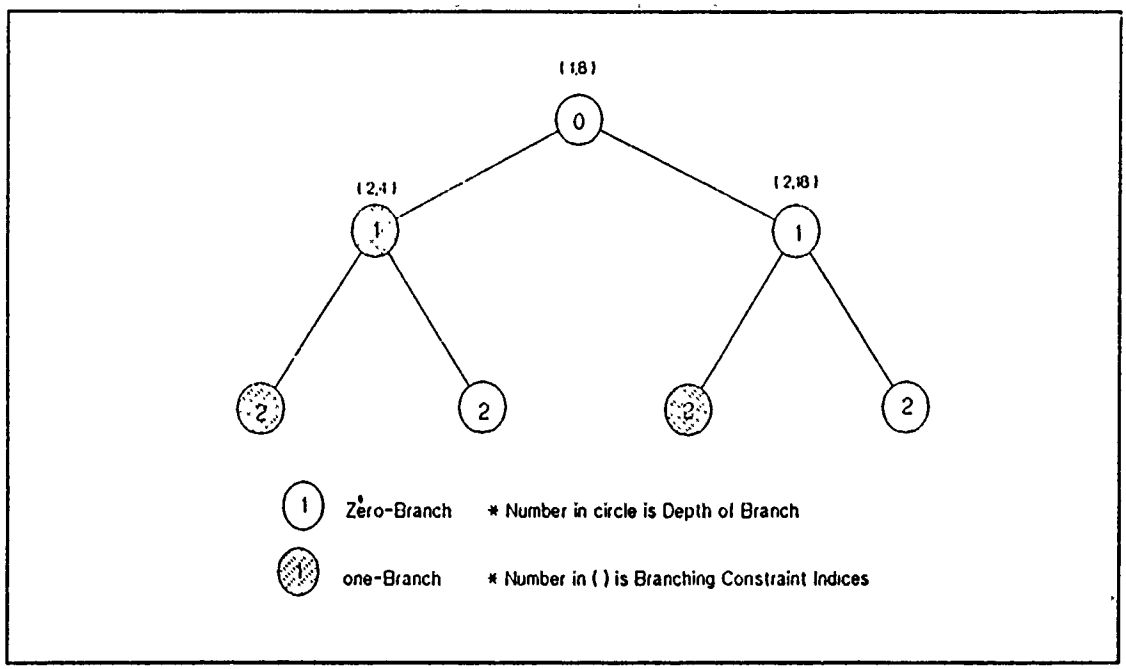


Figure 4. Constraint enumeration tree for SPD2X

required less CPU time to solve the problem, despite the fact that all subproblems were solved from scratch.

Figures 1 and 2 show the enumeration trees for problem T12 for variable branch and bound and for constraint branch and bound, respectively. Figures 3 and 4 are analogous for problem SPD2X. These trees are typical of the problems tested. It can be seen clearly that the enumeration trees for the constraint branch-and-bound method are more balanced than for the variable branch-and-bound method.

D. RECOMMENDATIONS

The program was tested using only small sample problems, but most of the real-world set partitioning problems are very large, usually including several hundred constraints and several thousand variables. Testing should be extended to real-world problems to ensure that the dramatic improvements seen here hold in practice. Additionally, a more efficient implementation of the constraint branch-and-bound method would bring out its full potential. Finally, different branching and constraint selection procedures should be tested for effectiveness.

E. CONCLUSIONS

The constraint branch-and-bound method to solve the set partitioning problems appears to be significantly more efficient than the conventional variable branch-and-bound method. Even using an inefficient implementation the constraint branch-and bound-method takes, on average, only 70.0 % of the CPU time of the variable branch-and-bound method when executed on small test problems. On average, the constraint branch-and-bound method had 59.3 % fewer nodes in its enumeration trees. Furthermore, the enumeration trees encountered are shallower and better balanced.

APPENDIX A. SAMPLE PROBLEMS WITH ENUMERATION TREES

A. SAMPLE PROBLEM JUL

```
MIN  5A + 4B + 3C + 2D + 4E + 3F + 3G + 2H + 2I
   + 3J + 3K + 4L + 2M + 2N + 3O + 4P + 4Q + 3R
   +  S + 4T + 2U + 2V + 3W + 3X + 5Y + 4Z + 3AA + 2AB + 4AC

 S.T.

 A+B+C+D+E+F+G+H+I                  = 1
 J+K+L+M+N+O+P                      = 1
 A+B+C+Q+R                          = 1
 J+K+L+M+N+O+F+S+T+U+V+W+X+Y        = 1
 C+D+E+F+G+K+N+O+S+T+U+V+Z+AA+AB    = 1
 E+F+H+I+L+N+P+T+V+W+X+Z+AA+AC      = 1
 B+G+O+Q+U+AB                       = 1
 C+F+I+M+P+R+V+X+Y+AA+AC            = 1

END

INT 29
```

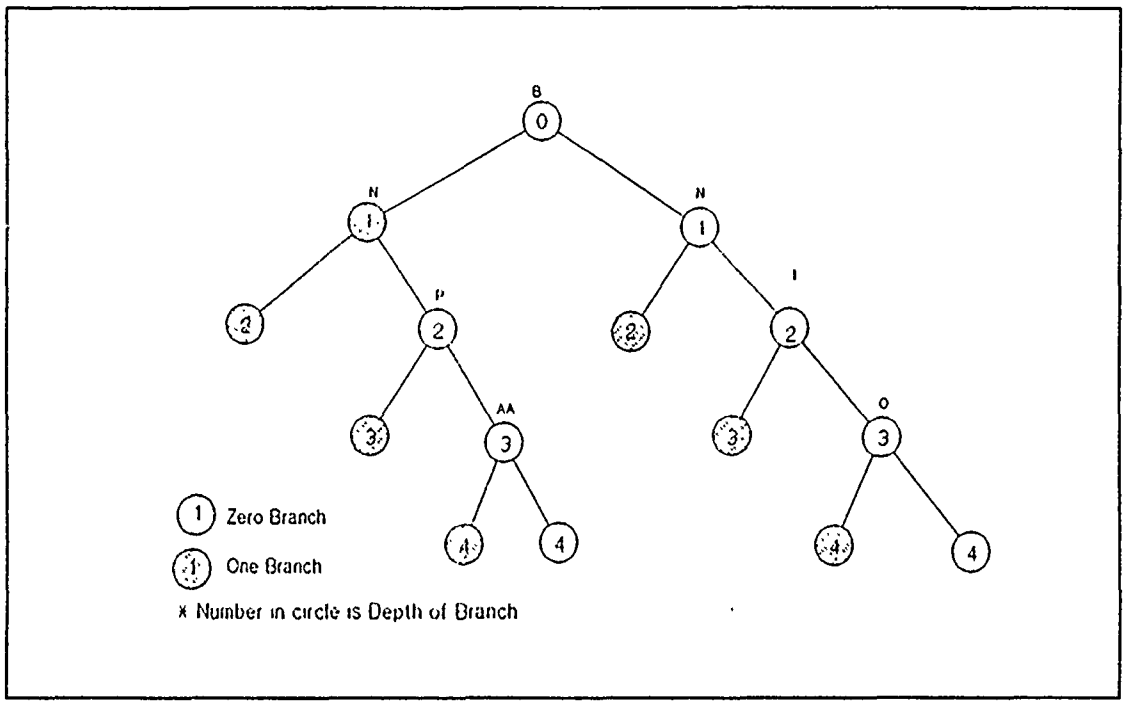


Figure 5. Variable enumeration tree for JUL

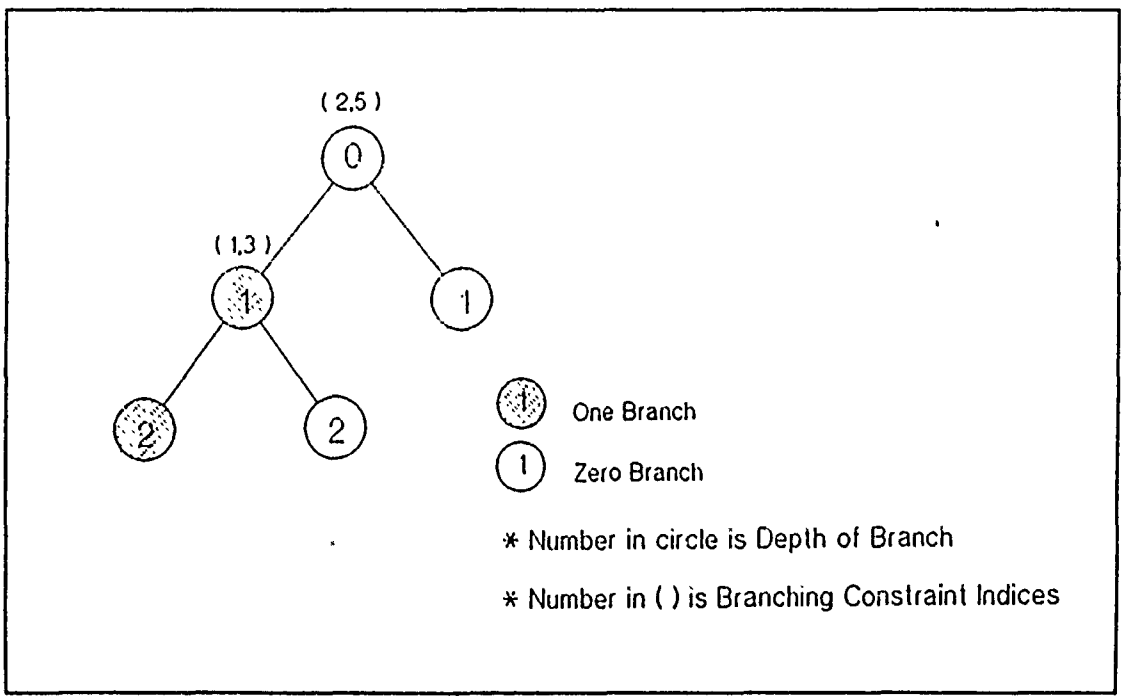


Figure 6. Constraint enumeration tree for JUL

B. SAMPLE PROBLEM AIR

```
MIN  2T1 +  2T2 +  2T3 +  3T4 +  3T5 +  2T6 +  2T7 +  3T8 + 3T9
  + 3T10 + 4T11 + 3T12 + 3T13 + 4T14 + 2T15 + 4T16 + 3T17 + 4T18
  + 2T19 + 4T20 + 4T21 + 5T22 + 4T23 + 5T24 + 5T25 + 6T26 + 5T27
  + 4T28 + 5T29 + 4T30 + 5T31 + 5T32 + 2T33 + 3T34 + 4T35 + 5T36 + 5T37

S.T

T1+T11+T16+T25+T26+T29+T34                          = 1
T2+T12+T13+T19+T27+T30+T35                          = 1
T3+T14+T15+T29+T30                                  = 1
T4+T16+T20+T22+T25+T27+T31+T34+T36                  = 1
T5+T17+T18+T31+T32                                  = 1
T6+T12+T17+T33                                      = 1
T7+T15+T19+T21+T23+T26+T28+T30+T32+T33+T35+T37      = 1
T8+T13+T18+T20+T21+T27+T28+T31+T32+T34              = 1
T9+T11+T22+T23+T24+T25+T26+T35+T36+T37              = 1
T10+T24+T36+T37                                     = 1
T2+T14+T16+T23+T24+T26+T31+T35                      = 1
T4+T13+T20+T25+T28+T30+T32+T36                      = 1
T7+T12+T15+T19+T21+T27+T33+T35+T37                  = 1

END

INT 37
```

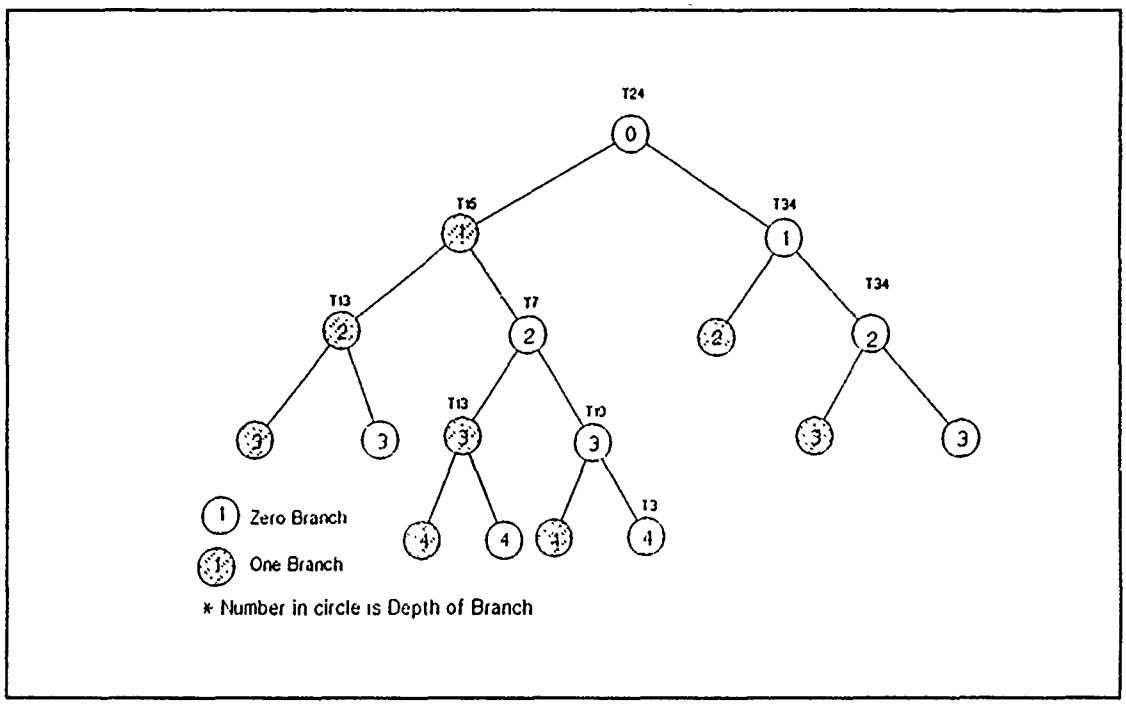


Figure 7. Variable enumeration tree for AIR

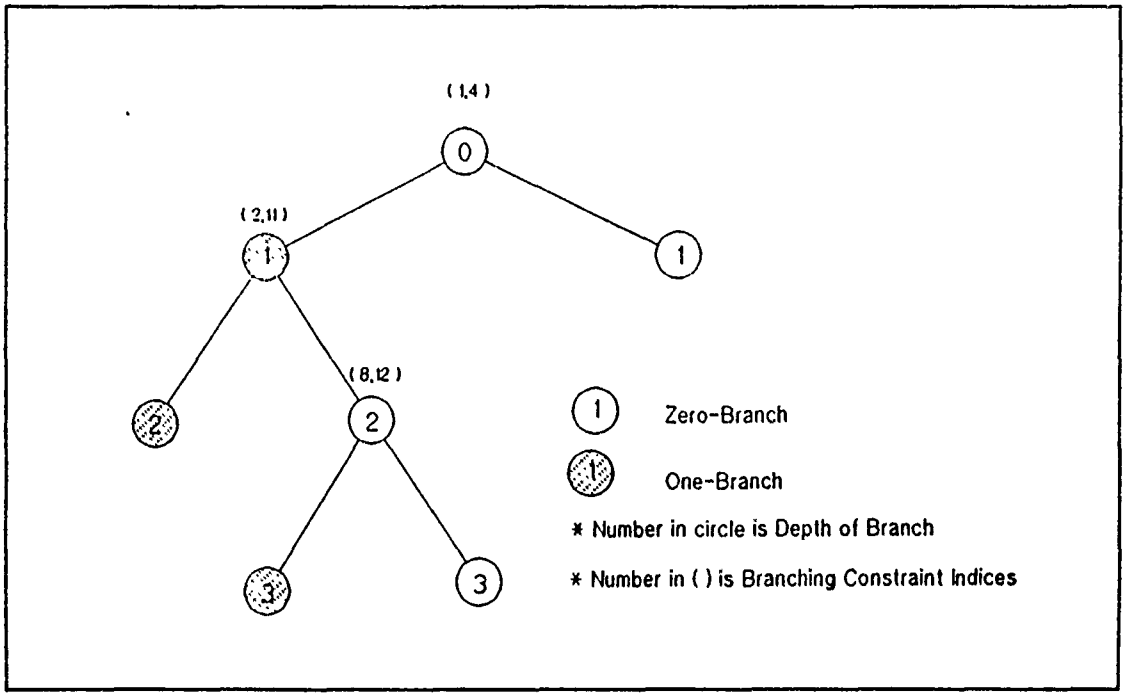


Figure 8. Constraint enumeration tree for AIR

C. SAMPLE PROBLEM DON

```
 MIN  X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10
    + Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
    + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10
    + Z1 + Z2 + Z3 + Z4 + Z5 + Z6 + Z7 + Z8 + Z9 + Z10

S.T.

   X1+X2+X3+X4+X5+X6+X7+X8          =  1
   X9+X10+Y1+Y2+Y3+Y4+Y5+Y6+Y9      =  1
   Y7+Y8+Y9+Y10+T1+T4+T6            =  1
   T4+T5+T6+T8+T9+Z1+Z3             =  1
   T7+T9+T10+Z1+Z2+Z3+Z4            =  1
   Z1+Z5+Z6+Z7+Z8+Z9+Z10            =  1
   X2+X6+X7+Z1+Z4+Z5+Z7+Z9          =  1
   X4+X8+Y6+Y7+Y8+Y10+T1+T2         =  1
   X4+X6+X7+X8+Y2+Y4+Y6             =  1
   Y1+Y2+Y4+Y8+T4+T6+T7+T10         =  1
   X5+X8+X10+Y1+Y4+Y8+Y10           =  1
   Y7+Y10+T2+T6+T9+Z2+Z3+Z4+Z6      =  1
   Y1+Y3+Y4+Y7+Y10+T5+T6+T9         =  1
   X6+X9+Y2+Y4+Y7+Y10+Z5+Z10        =  1
   T1+T3+T5+T9+T10+Z3+Z6            =  1
   Y9+T2+T5+T7+T8+Z2+Z3+Z4          =  1
   Y4+Y7+Y8+T2+T3+T4+T9+T10+Z2      =  1
   T2+T6+T7+T8+T10+Z1+Z4+Z6+Z8      =  1
   X2+X3+X4+X5+Z1+Z6+Z7+Z8          =  1
   X8+Y2+Y5+Y7+Y8+Y9+T2+T3          =  1
END

INT 40
```

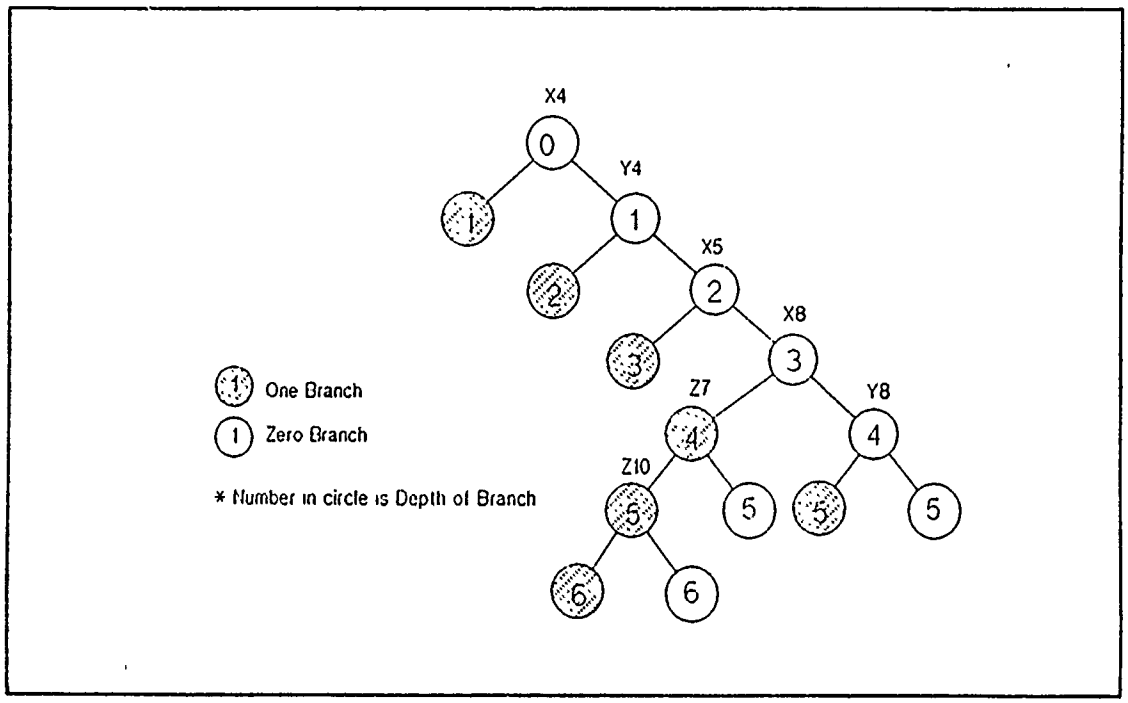


Figure 9. Variable enumeration tree for DON

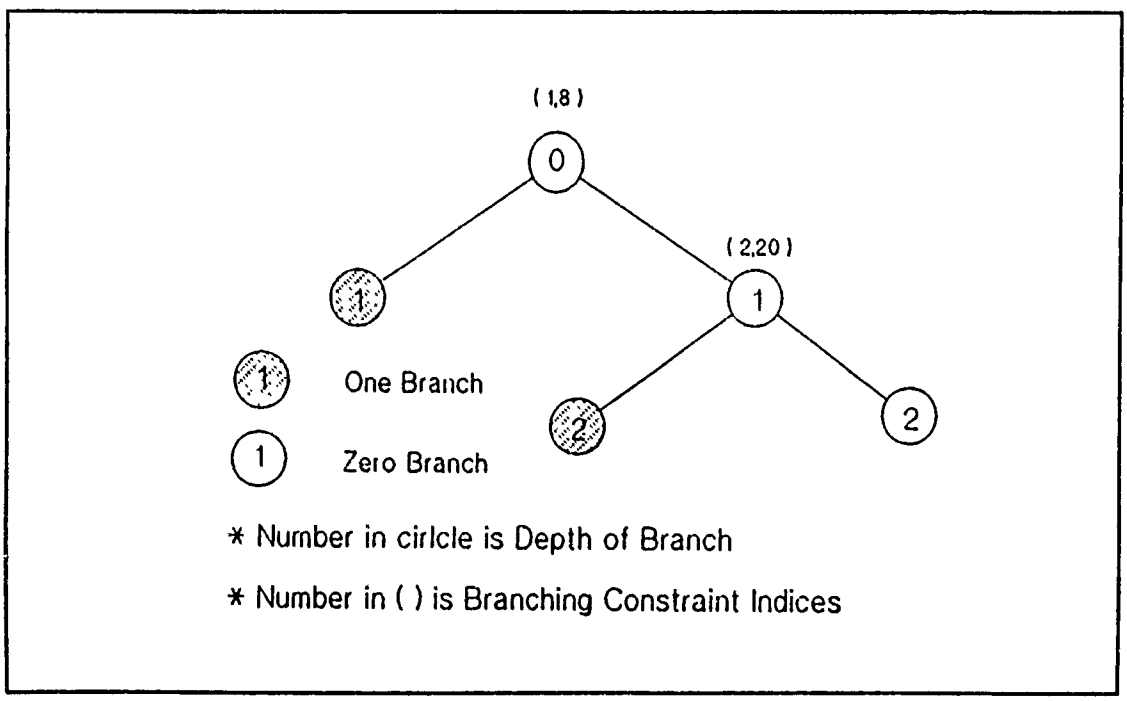


Figure 10. Constraint enumeration tree for DON

D. SAMPLE PROBLEM T12

```
MIN   X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10
    + Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
    + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10
    + Z1 + Z2 + Z3 + Z4 + Z5

S.T.

 X1+X3+X4+X5+X6+X8                          = 1
 Y9+Y10+T3+T4+T6+T10+Z1+Z2+Z3               = 1
 X1+X5+X6+X9+T4+T6+T9+Z1+Z3+Z4              = 1
 X2+X6+X7+X10+Y4+Y8+Y9+T3+T5                = 1
 X1+X3+X4+X6+Y6+Z1+Z4                       = 1
 X8+X9+X10+Y1+Y3+Y4+Y8+Y10+Z5               = 1
 Y1+Y4+Y5+Y6+Y7+Y10+T1+T3+T4+T10            = 1
 Y5+Y8+T1+T2+T3+T5+T6+T7+Z2+Z3+Z4           = 1
 X6+X7+X9+Y7+Y8+T1+T10+Z1+Z2+Z4+Z5          = 1
 X2+X7+Y1+Y5+Y6+Y9+T6+T9+Z2+Z3              = 1
 X10+Y3+Y4+Y6+Y7+Y8+Z2+Z3+Z4+Z5             = 1
 X4+X9+Y4+Y7+Y8+Y9+Y10+T2+T3+T10+Z1         = 1
 X6+X7+X9+Y8+T2+T3+T4+T7+Z1+Z3+Z5           = 1
 X9+Y1+Y2+Y4+T2+T4+T5+T10                   = 1
 Y2+Y8+Y10+T1+T3+T8+Z2+Z3+Z5                = 1

END

INT 35
```

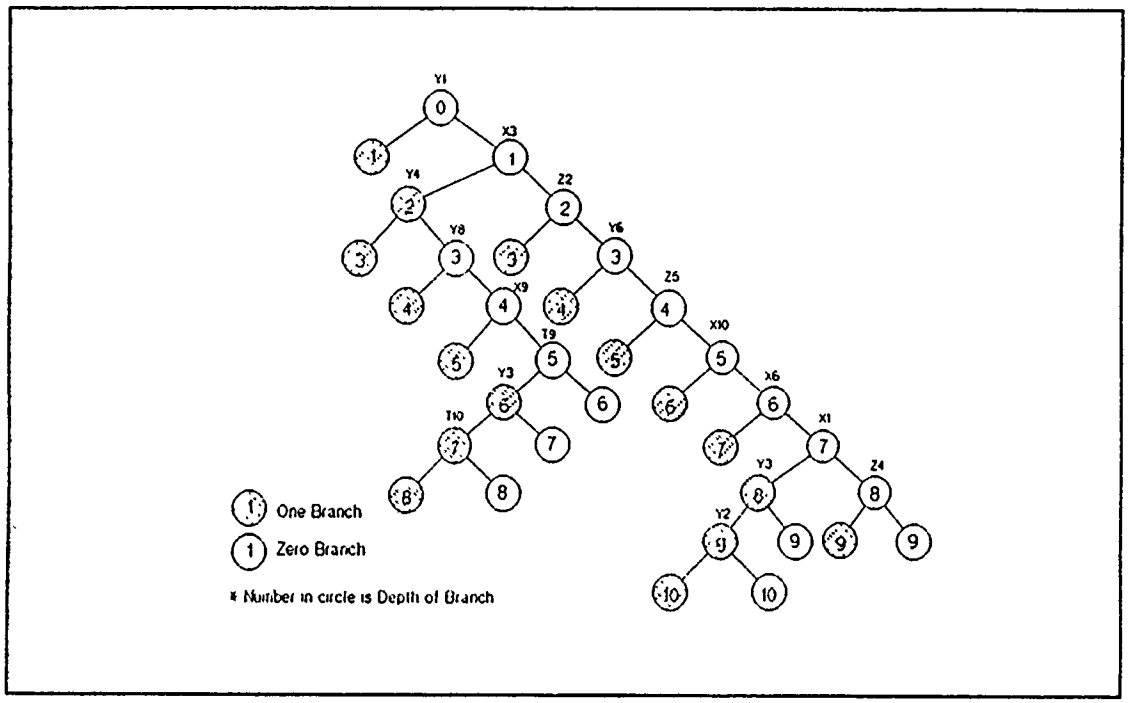


Figure 11. Variable enumeration tree for T12

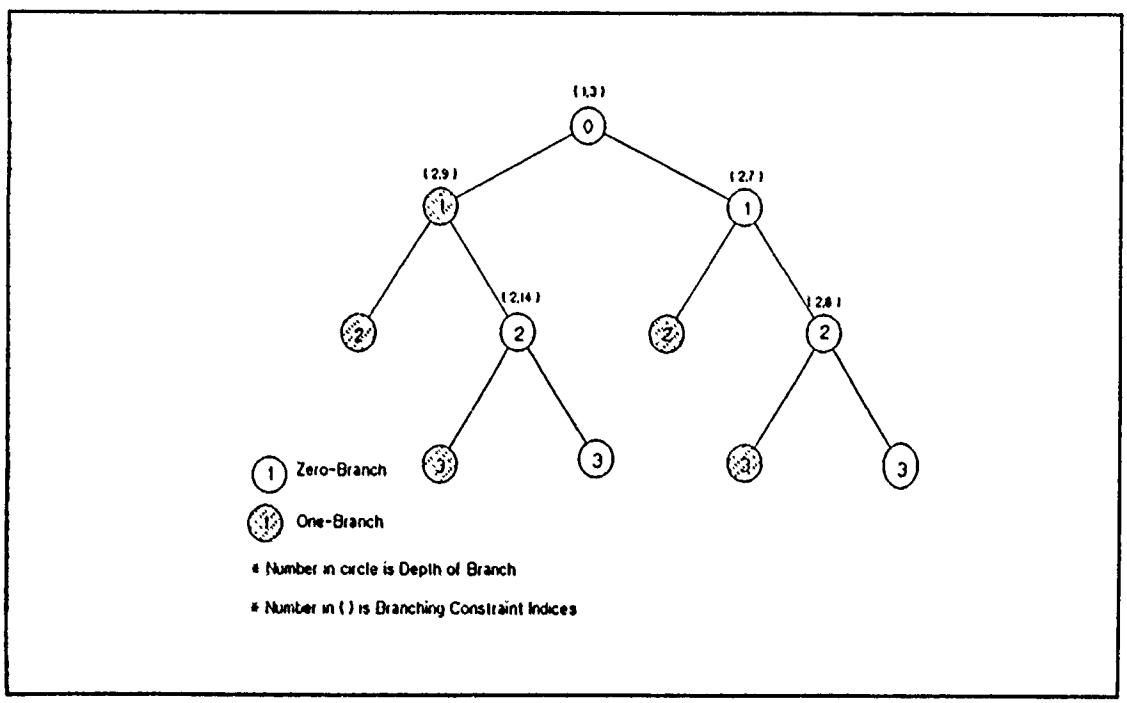


Figure 12. Constraint enumeration tree for T12

E. SAMPLE PROBLEM D3

```
MIN   X1 +  X2 +  X3 +  X4 + 3X5 + X6 + X7 +  X8 +  X9 + X10
    + Y1 + 5Y2 +  Y3 +  Y4 + 7Y5 + Y6 + Y7 + 4Y8 +  Y9 + Y10
    + T1 + 3T2 +  T3 +  T4 +  T5 + T6 + T7  + T8 +  T9 + T10
    + Z1 +  Z2 +  Z3 + 2Z4 +  Z5 + Z6 + Z7 +  Z8 + 4Z9 + Z10
    + U1 + 5U2 + 3U3 +  U4 + 3U5

S.T.

 X1+X2+X3+X4+X5+X6+X7+Z8                          =  1
 X8+X9+X10+Y1+Y2+Y3+Y4+Y5+Y6                      =  1
 Y7+Y8+Y9+Y10+T1+T2+T3+T4+T5                      =  1
 T6+T7+T8+T9+T10+Z1+Z2+Z3+Z4+Z5+Z6+Z7+Z8          =  1
 T6+T7+T8+T9+T10+U1+U2+U3+U4+U5                   =  1
 Z2+Z7+Z8+U1+U2+U5+Z9                             =  1
 X3+X4+X6+T7+Z1+Z4+U1+U4                          =  1
 Y1+Y3+Y4+Y6+Y8+Y10+T2+T8+T9+T10                  =  1
 X1+X4+X6+X7+X8+T7+Z1+Z4+Z5+Z6+Z10                =  1
 T3+T5+T6+T7+T9+T10+Z1+Z2+Z3+Z4                   =  1
 T4+T7+T8+T9+T10+Z1+Z2+Z3+Z4+Z5                   =  1
 X4+X7+X9+X10+Y1+Y4+Y9+T2+T4+U1+U3                =  1
 X2+X7+T6+T9+Z2+Z3+Z4+Z5                          =  1
 X10+T3+T4+T5+T6+Z2+Z3+Z4+Z6+Z8                   =  1
 Y4+Y7+Y10+T2+T9+Z7+Z10+U1+U3                     =  1
 X1+X4+X10+Y2+Y10+Z1+Z3+Z5+U1+U5                  =  1
 Y1+Y3+Y6+Y7+Y9+Z2+Z3+Z4+Z5+Z7+Z8                 =  1
 X6+X7+T2+T3+T4+T7+T8+T9+T10+Z1+Z3+Z9             =  1
 T5+T6+T7+T8+T10+Z1+Z4+Z6                         =  1
 T1+T3+T8+T9+T10+Z1+Z4+Z8                         =  1

 END

INT 45
```

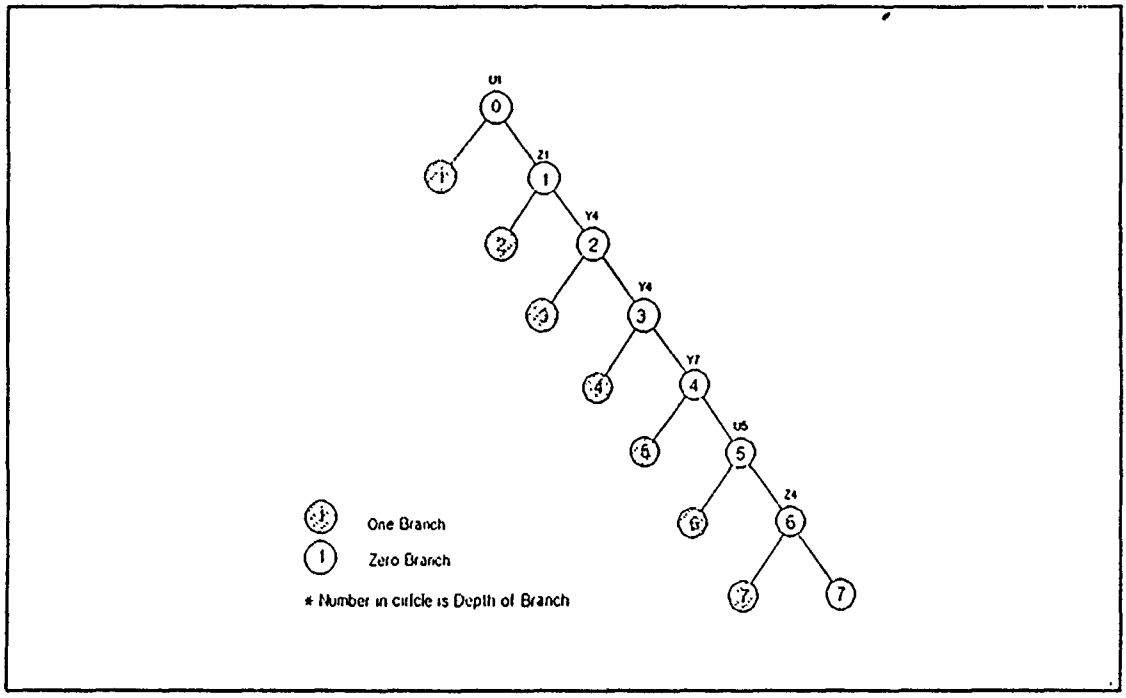


Figure 13. Variable enumeration tree for D3

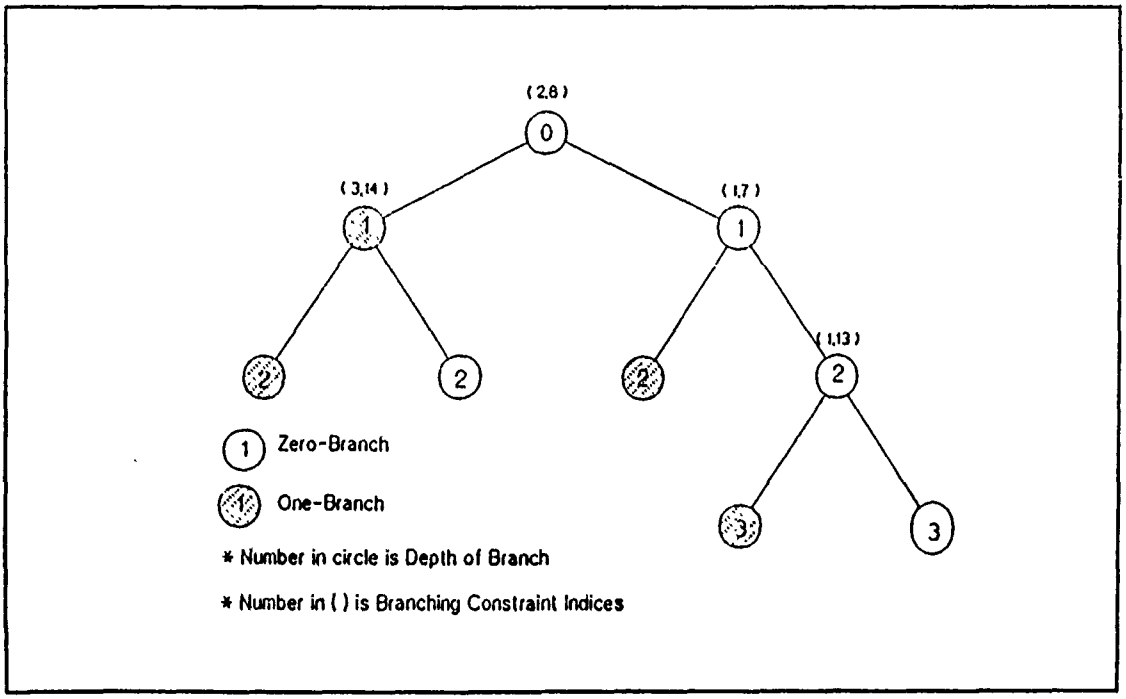


Figure 14. Constraint enumeration tree for D3

F. SAMPLE PROBLEM SPD2X

```
 MIN  X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10
    + Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
    + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10
    + Z1 + Z2 + Z3 + Z4 + Z5 + Z6 + Z7 + Z8 + Z9 + Z10
    + 90AA + 90AB + 90AC + 90AD + 90AE + 90AF + 90AG
    + 90AH + 90AI + 90AJ + 90AK + 90AL + 90AM + 90AN
    + 90AO + 90AP + 90AQ + 90AR + 90AS + 90AT

S.T.

  X1 + X3 + X4 + X5 + X6 + X8 + AA          =  1
  T4 + T6 + T10 + Z1 + Z2 + Z3 + Z7 + AB     =  1
  Y8 + Y9 + Z2 + Z6 + Z10 + Z9 + AC          =  1
  Y1 + Y2 + Y3 + Z1 + Z3 + Z4 + AD           =  1
  X6 + X10 + Z4 + Z6 + Z7 + Z9 + AE          =  1
  X4 + X6 + Y6 + Y7 + Y9 + AF                =  1
  X2 + X8 + X9 + X10 + Y1 + Y3 + AG          =  1
  X4 + X6 + X7 + X8 + Y1 + Y4 + AH           =  1
  T9 + T10 + Z1 + Z2 + Z3 + Z4 + AI          =  1
  Y3 + Y4 + Y7 + Y8 + T9 + T10 + AJ          =  1
  X7 + X9 + X10 + Y1 + Y4 + Y9 + AK          =  1
  T6 + T9 + Z2 + Z3 + Z4 + Z5 + AL           =  1
  Y3 + Y4 + Y6 + T5 + T6 + AM                =  1
  Y4 + Y7 + Y10 + Z7 + Z10 + AN              =  1
  T1 + T3 + T5 + T9 + T10 + AO               =  1
  T5 + T7 + T8 + Z2 + Z3 + Z4 + AP           =  1
  Y7 + Y8 + T2 + T3 + T4 + T9 + T10 + AQ     =  1
  T6 + T7 + T8 + T10 + Z1 + Z4 + Z6 + AR     =  1
  X2 + X3 + X4 + X5 + Z1 + Z4 + Z8 + AS      =  1
  Y2 + Y5 + Y7 + Y8 + Y9 + AT                =  1

 END

INT 60
```

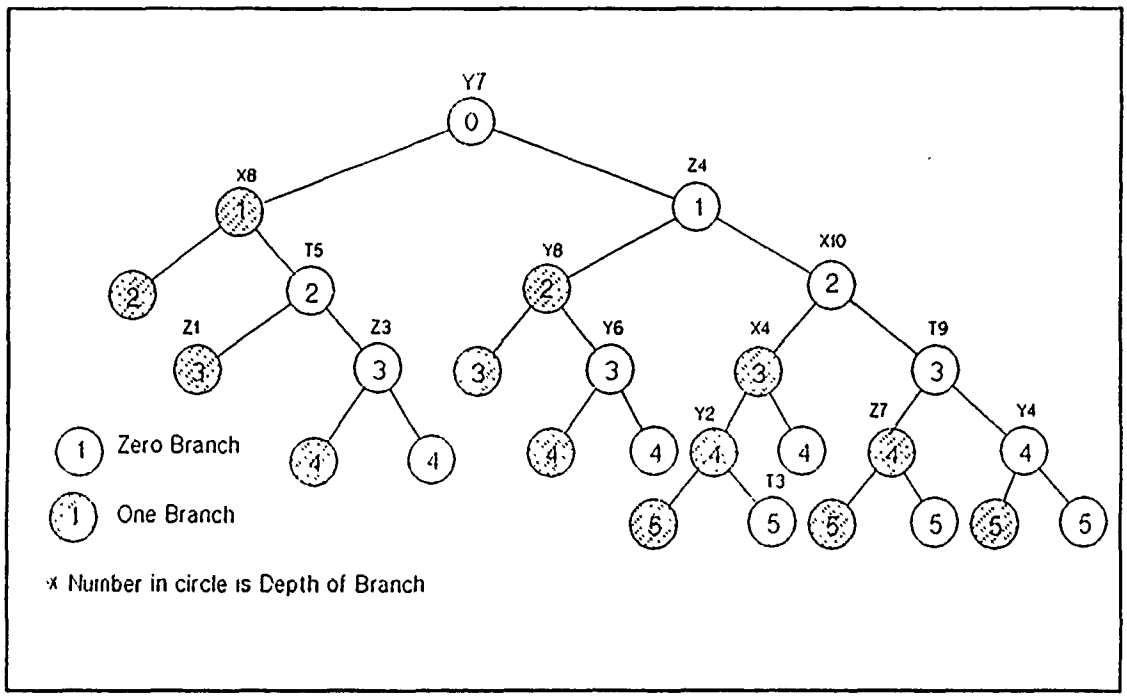


Figure 15. Variable enumeration tree for SPD2X

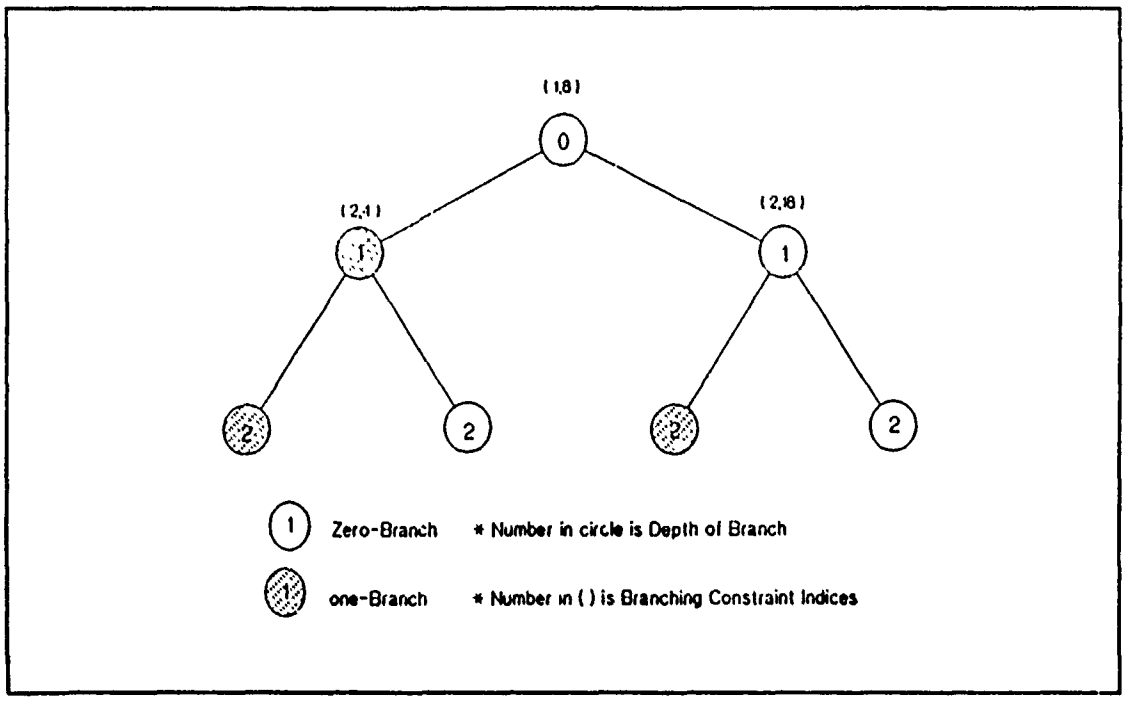


Figure 16. Constraint enumeration tree for SPD2X

G. SAMPLE PROBLEM D3X

```
MIN   X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10
    + Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
    + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10
    + Z1 + Z2 + Z3 + Z4 + Z5 + Z6 + Z7 + Z8 + Z9 + Z10
    + U1 + U2 + U3 + U4 + U5
    + 90AA + 90AB + 90AC + 90AD + 90AE + 90AF + 90AG + 90AH + 90AI
    + 90AJ + 90AK + 90AL + 90AM + 90AN + 90AO + 90AP + 90AQ + 90AR
    + 90AS + 90AT

S.T.

 X1 + X2 +X3 + X4 + X5 + X6 + X7 + AA                          = 1
 X8 + X9 + X10 + Y1 + Y2 + Y3 + Y4 + Y5 + AB                    = 1
 Y6 + Y7 + Y8 + Y9 + Y10 + T1 + T2 + T3 + T4  +AC               = 1
 X3 + X8 +T5 + T6 + T7 + T8 + T9 + T10 + Z4 + AD                = 1
 Z2 + Z4 + Z5 + Z6 +   Z7 + AE                                  = 1
 Z5 + Z8 + Z10 + U1 + U3 + U4 +  AF                             = 1
 X4 + X7 + X9 + T5 + Z1 + Z6 + U1 + U4 +AG                      = 1
 Y1 + Y5 +  Y8 + Y9  + T3 + T5 + T6 + T9  +AH                   = 1
 X3 + X8 +  X9 + Z2 + Z4 + Z6 +  Z10 +AI                        = 1
 Y10 + T2 + T3 + T7 + T9  + Z1 + Z2 + Z7 + Z4+AJ                 = 1
 T2 + T3 +  T7 + T10 + Z2 + Z4 + Z5 +AK                         = 1
 X3 + X6 + X8  + X10 + Y5 + Y7 +  Y9 + T2 + T4 + U1 + U4 +AL    = 1
 Y1 + Y3 + Y5 + Y6 + T7 + T9 + Z1 + Z3 +  Z7 +AM                 = 1
 T1+ T3 + T6 + Z2 + Z3 + Z6 +  Z8 +AN                           = 1
 Y2 + Y4 + Y7 + Y9  + T4 + Z7 + Z9  + U1 + AO                   = 1
 X2 + X4 + X8 +  Z2 + Z3 + Z5 + U1 + U5 +AP                     = 1
 Y1 + Y3 + Y6 + Y7 + Y9 + Z2 + Z3 + Z4 + Z5 + Z7 + Z8+AQ        = 1
 T2 + T3 + T4 + T7 + T8 + T9 + T10 + Z1 + Z3 + Z9   +AR         = 1
 T3+ T5 + T6 + T9 + T8 + Z1 + Z4 + Z5 + U5 + AS                 = 1
 T5 + T6 + T9 + T10 + Z1 + Z4 + Z7 +AT                          = 1

END

INT 65
```

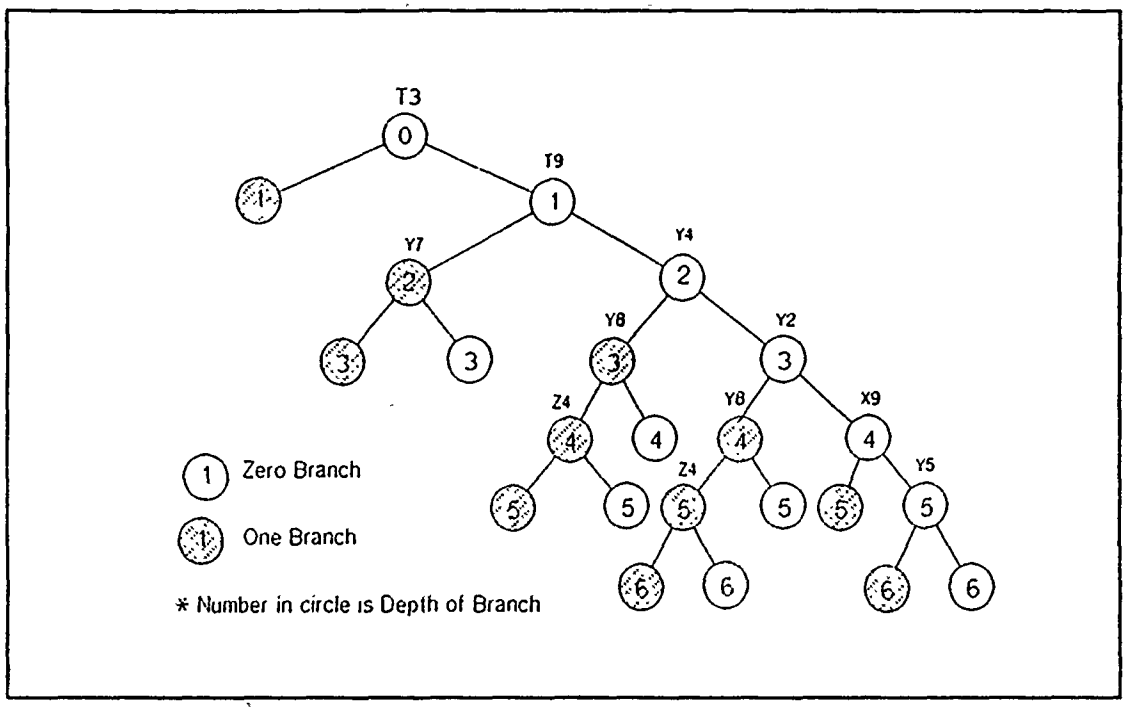


Figure 17. Variable enumeration tree for D3X

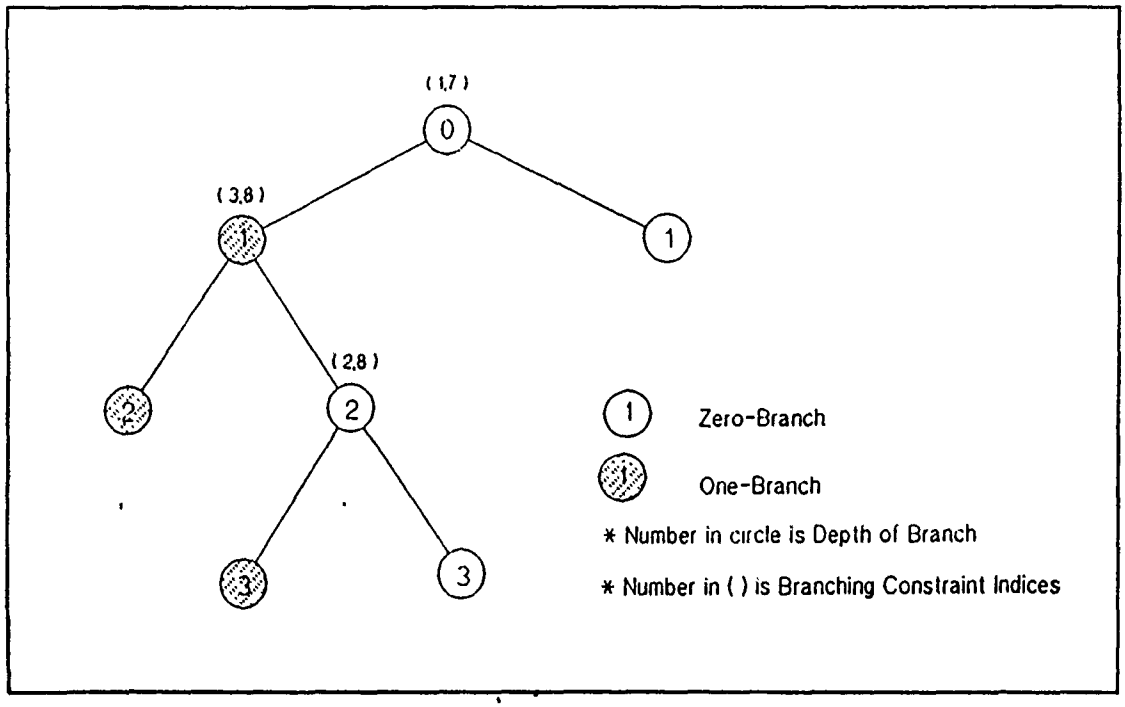


Figure 18. Constraint enumeration tree for D3X

H. SAMPLE PROBLEM D4

```
MIN   X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10
    + Y1 + Y2 + Y3 + Y4 + Y5 + Y6 + Y7 + Y8 + Y9 + Y10
    + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10
    + Z1 + Z2 + Z3 + Z4 + Z5 + Z6 + Z7 + Z8 + Z9 + Z10
    + U1 + U2 + U3 + U4 + U5

S.T.

  X1 + X2 + X4 + X5 + X6 + X7 + Z8                    =  1
  X8 + X9 + X10 + Y1 + Y2 + Y3 + Y4 + Y5 + Y6         =  1
  Y7 + Y8 + Y9 + Y10 + T1 + T2 + T3 + T4 + T5         =  1
  T6 + T7 + T8 + T9 + T10 + Z1 + Z2 + Z3 + Z4 + Z5    =  1
  T6 + T7 + T8 + T9 + T10 + U1 + U2 + U4 + U5         =  1
  Z2 + Z7 + Z8 + U1 + U2 + U3 + Z5                    =  1
  X4 + X6 + T7 + Z1 + Z4 + U1 + U4                    =  1
  Y1 + Y3 + Y4 + Y8 + Y10 + T2 + T8 + T9              =  1
  T5 + T6 + T7 + T9 + Z1 + Z2 + Z3 + Z4               =  1
  T4 + T7 + T8 + T9 + Z1 + Z2 + Z3 + Z4 + Z5          =  1
  X7 + X9 + X10 + Y1 + Y4 + Y9 + T4 + U1 + U3         =  1
  X2 + X7 + T6 + T9 + Z2 + Z3 + Z4 + Z5               =  1
  X10 + T3 + T4 + T6 + Z2 + Z3 + Z4 + Z6 + Z8         =  1
  Y4 + Y7 + Y10 + T2 + T9 + Z7 + Z10 + U1 + U3        =  1
  X1 + X4 + X10 + Y2 + Y10 + Z1 + Z3 + Z5 + U1 + U5   =  1
  Y6 + Y7 + Y9 + Z2 + Z3 + Z4 + Z5 + Z7 + Z8          =  1
  T3 + T4 + T7 + T8 + T9 + T10 + Z1 + Z3 + Z9         =  1
  T5 + T6 + T7 + T8 + T10 + Z1 + Z4 + Z6              =  1
  T1 + T3 + T8 + T9 + T10 + Z1 + Z4 + Z8              =  1
  Y1 + Y2 + Y5 + Y7 + Y8 + Y9 + T3 + T4 + T8          =  1

END

INT 45
```

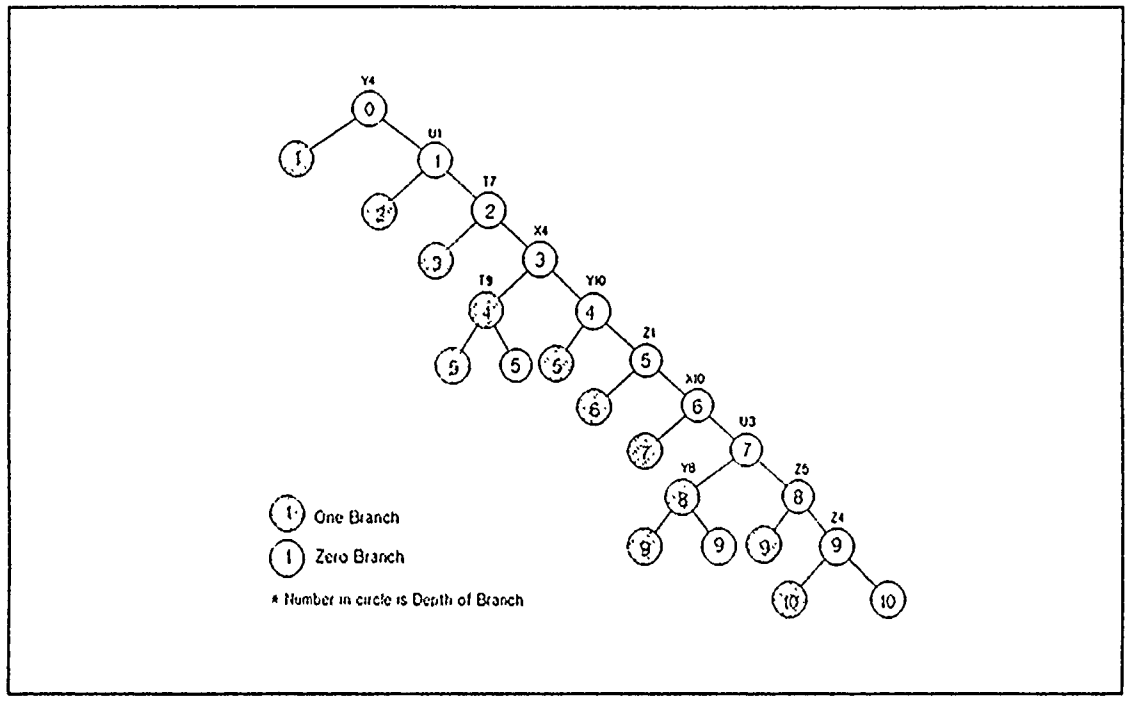


Figure 19. Variable enumeration tree for D4

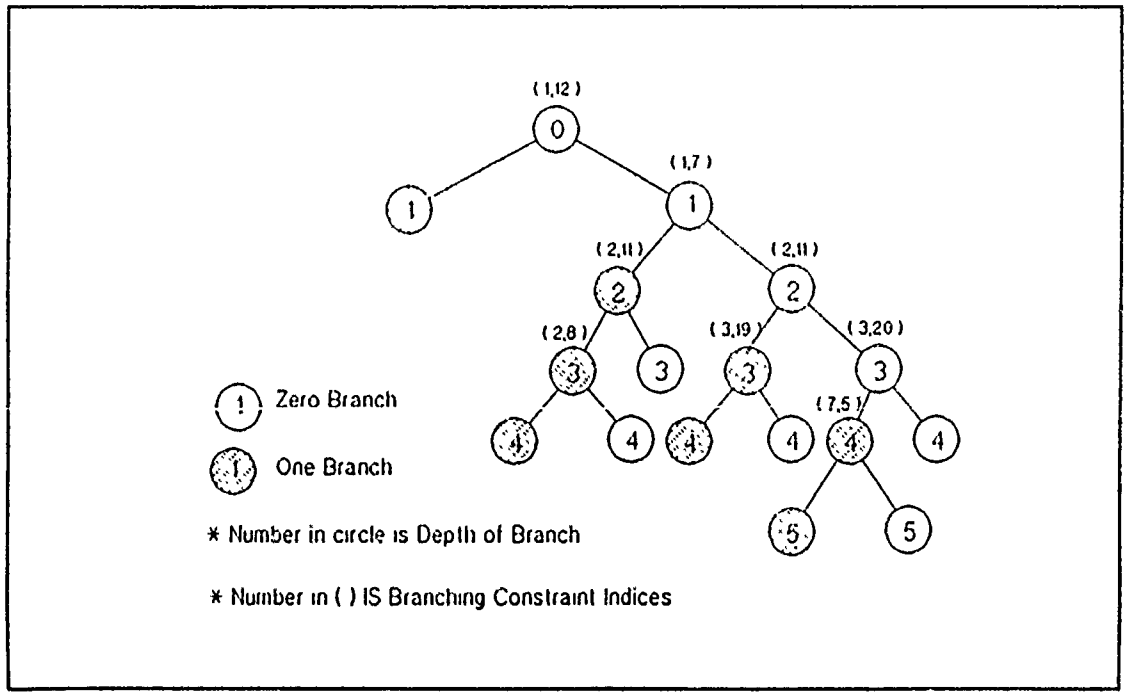


Figure 20. Constraint enumeration tree for D4

LIST OF REFERENCES

1. Garfinkel, R. S. and Nemhauser, G. L., *Integer Programming*, Wiley-Interscience, pp. 298-322, 1972.

2. Arabeyre, J. P., Fearnley, J., Steiger, F. C. and Teather, W., *The air crew scheduling problem; A survey*, Transport Science, Vol. 3, No. 2, pp. 140-163, 1969.

3. Marsten, R. E., and Shepardson, F., *Exact solution of crew scheduling problems using the set partitioning model: recent successful application*, Network 11, pp. 165-177, 1981.

4. Clarke, G. and Wright, S. W., *Scheduling of vehicles from a central depot to a number of delivery points*, Operations Research, Vol. 12, No. 4, pp. 568-581, 1964.

5. Garfinkel, R. S. AND Nemhauser, G. L, *Optimal political districting implicit enumeration techniques*, Management Science, Vol. 16, No. 8, pp. 495-508, 1970.

6. Foster, B. A. and Ryan, D. M., *An integer programming approach to the vehicle scheduling problem*, Operations Research Quarterly 27, pp. 367-384, 1976.

7. Smith, B. M., *Bus crew scheduling using mathematical programming*, PH.D. Thesis, University of Leeds, 1986.

8. Balas, E. and Padberg, M. W., *Set partitioning: a survey,* SIAM Review 18, pp. 710-761, 1976.

9. Etcheberry, J., *The set-covering problem: a new implicit enumeration algorithm,* Operations Research Vol. 25, No. 5, pp. 760-772, 1977.

10. Davis, R. E., Kendrick, D. A., and Weitzman, M., *A branch and bound algorithm for zero-one mixed integer programming problems,* Operations Research Vol. 19, No. 4, pp. 1036-1044, 1971.

11. Little, J. D. C., Murty, K. G., Sweeney, D. W., and Karel, C., *An algorithm for the traveling salesman problem,* Operations Research Vol. 11, pp. 972-989, 1963.

12. Hey, A. M., *An algorithm for the set covering problem,* PH.D. Thesis, University of London, 1981.

13. Falkner, J. C., and Ryan, D. M., *A bus crew scheduling system using a set partitioning model,* Asia-Pacific Journal of Operations Research 4(1), 1987.

14. Marsten R. E., *An algorithm for large set partitioning problems,* Management Science, Vol. 20, No. 5, pp. 774-787, 1974.

15. Brown, G. G. and Graves, G. W., *XS mathematical programming system,* Perpetual working paper, (C. 1983).

16. Falkner. J. C., *Bus crew scheduling and the set partitioning model,* PH.D. Thesis, University of Auckland, New Zealand, November 1988.

17. Linus Schrage *Linear, integer, and quadratic programming with LINDO,* The Scientific Press, pp. 70-73, 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 55 Department of Operations Research Naval Postgraduate School	2
4.	Professor R. Kevin Wood, Code 55Wd Department of Operations Research Naval Postgraduate School	6
5.	Professor Gerald G. Brown Code 55Bw Department of Operations Research Naval Postgraduate School	4
6.	System Analysis Department ROK Army Headquarters Nonsan-Gun, Duma-Myen, Bunam-ri, CPO Box #2 Chungchungnam-Do, 320-919 Republic of Korea	4
7.	Park, Rae Yun SMC #2329 Naval Postgraduate School Monterey, Ca 93943	1
8.	Lee, Dong Keun SMC #2294 Naval Postgraduate School Monterey, Ca 93943	1
9.	Kim, Sook Han SMC #2318 Naval Postgraduate School Monterey, Ca 93943	1
10.	Lee, Kyung Taek SMC #2990 Naval Postgraduate School Monterey, Ca 93943	1

10. Choi, Byung Gook — 1
SMC #1587
Naval Postgraduate School
Monterey, Ca 93943

11. Ryoo, Moo Bong — 2
Chungchungbuk-Do, Jungwon-Gun, Sancheok-Myen,
Youngdeok-Ri, Hayoung
Republic of Korea